Parallel Algorithms for Some Problems on Perfect Graphs

by
NAINENI MALAHAL RAO

DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING
INDIAN INSTITUTE OF TECHNOLOGY KANPUR
January 1994

Parallel Algorithms for some Problems on Perfect Graphs

A thesis submitted
in partial fulfillment
of the requirements
for the degree of

Master of Technology

by

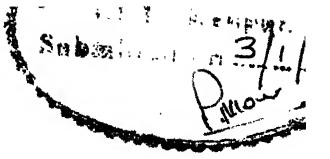
Naineni Malahal Rao

to the

Department of Computer Science and Engineering

Indian Institute of Technology, Kanpur

January, 1994



Certificate

It is certified that the work contained in the thesis titled *Parallel Algorithms for some Problems on Perfect Graphs*, by Naineni Malahal Rao, has been carried out under my supervision and that this work has not been submitted elsewhere for a degree.

January 1, 1993

Dr. Sanjeev Saxena
Dept. of Computer Sc. & Engg.
IIT Kanpur

Dedicated to the countless rishis
who taught self-abnegation to the whole world

Acknowledgments

First and foremost, I would like to express my deepest gratitude to my guide, Dr. Sanjeev Saxena, for the help he extended throughout the thesis. The two days he spent with me for taking laser printouts of this report will linger long in my memory.

I am greatly indebted to Dr. Ashish Mukhopadyaya for his excellent teaching, insight and inspiration. I am grateful to my parents, brother and sisters for their love and encouragement.

I owe a substantial debt to too many individuals. Saibal and S. V. Rao helped me in preparing this report. Many helped me when ever I am in needbut Shyam Sundar and Sai in particular top the list. There are others, countless in number, who accompanied me in the Hall V lawn and in playing Cricket, Volley-ball, Table Tennis and Carroms.

My sincere thanks to all.

Contents

List of Figures

Abstract

The following parallel algorithms are investigated in this thesis:

- A parallel algorithm for finding k-connected components of interval graphs in $O(k \log\log n)$ time with $\frac{n}{\log\log n}$ processors on a COMMON CRCW PRAM is given, assuming that the intervals are given in sorted order. Further, if the end points are integers then it can be implemented in $O(k \log^* n)$ time with $\frac{n}{\log^* n}$ processors on a Priority write PRAM. Previous algorithm for finding biconnected components took $O(\log n)$ time with $\frac{n}{\log n}$ processors on EREW model.

- Prefix computation problem for lists is generalised to trees; the generalised problem is solved in $O(\log n)$ time using $\frac{n}{\log n}$ processors on EREW PRAM.

- With the help of the prefix computation on trees, the time processor product of the algorithm for computing longest common sub-sequence [37] can be improved from $(n \log^3 n)$ to $(n \log^{2.5} n)$.

- Optimal $O(\log n)$ time parallel algorithms are presented for permutation representation and coloring of cographs, given its cotree representation. The previous algorithm for this problem requires $O(\log n)$ time with n processors on EREW PRAM model.

- The problem of finding OR of n bits is reduced to the problem of finding the cardinality of maximum independent set of a permutation graph of $n + 1$ nodes. This proves a lower bound of $\Omega(\log n)$ on time for finding the cardinality of a Maximum Independent Set of Permutation Graph on a Concurrent Read ~~Concurrent~~ Exclusive Write PRAM.

  The problem of computing Parity of n bits is reduced to the problem of finding the cardinality of maximum independent set of a permutation graphs of $2n$ nodes; thus

proving a lower bound of $\Omega(\frac{\log n}{\log \log n})$ (for the latter problem) on time on a Concurrent Read Concurrent Write PRAM, if only $n^{O(1)}$ processors are to be used.

Chapter 1

Introduction

Graph theory was introduced by Euler, one of the greatest mathematicians of all time, to solve Königsberg bridge problem. Since then graph theory has been used in innumerable number of fields— its rediscovery, by Kirchhoff for analysing electrical networks and by Cayley for enumeration of organic chemical isomers, are some examples. A wide variety of applications of graphs are discussed in [20].

1.1 Graphs

There is no standard terminology in graph theory. For example in [27] the author calls *points* for vertices of the graph and *lines* for the edges of the graph. Hence the graph theoretical terminology used in this thesis is first explained.

- *Graph:* Graph[1] G consists of a finite set of vertices and a finite set of edges connecting the vertices. The graph itself is denoted by $G(V, E)$ where V denotes the set of vertices and E denotes the set of edges. If edge e_i connects node u and node v then the edge e_i is represented by (u, v) or (v, u).
- *Complete graph:* A graph with $\frac{n(n-1)}{2}$ edges, i.e. one in which every vertex is connected to every other vertex is called a complete graph. A complete graph of n vertices is represented by K_n.

[1]Throughout this thesis a 'graph' means a simple undirected graph.

- *Vertex induced subgraph:* Given a subset $A \subseteq V$ of the vertices then the subgraph of $G(V, E)$ induced by A is defined as the graph $H(A, E')$ where $E' \subseteq E$ and contains only edges from $A \times A$. It is represented by $G([A])$.

- *Edge induced subgraph:* Given a subset $F \subseteq E$ of the edges then the subgraph of $G(V, E)$ induced by F is defined as the graph $\Psi(A', F)$ where A' is the set of end vertices of edges in F; i.e. $v \in A'$ iff there exists a node u such that $(u, v) \in F$.

- *Clique:* A node induced subgraph, $G([A])$ of $G(V, E)$ is a clique if and only if $G([A])$ is isomorphic[2] to the complete graph with $\mid A \mid$ vertices.

- *Maximum clique:* A clique of maximum cardinality is called maximum clique. The cardinality of a maximum clique is denoted by $\omega(G)$; it is called the *clique number* of G.

- *Clique cover of size k:* A clique cover of size k is a partition[3] $V = A_1 \cup A_2 \cup \ldots \cup A_k$ such that each A_i is a clique. The size of a smallest clique cover is denoted by $\kappa(G)$; it is called the *clique cover number* of G.

- *Independent set:* A subset $A \subseteq V$ of the vertices is called an independent set if and only if the subgraph induced by vertices of A is a null[4] graph. The cardinality of a maximum independent set is denoted by $\alpha(G)$; it is also called the *stability number* of G.

- *Coloring:* Coloring is a process of giving colors (integers) to vertices of the graph.

- *Proper coloring:* A graph is said to be properly colored if no two adjacent vertices have the same color. Minimum number of colors needed to properly color the graph is denoted by $\chi(G)$; it is called the *chromatic number* of the graph.

---

[2]Graphs G_1 and G_2 are isomorphic to each other iff there is a one-one correspondence between their vertices with adjacency property.

[3]$A_i \cap A_j = \phi$ for $i \neq j$ and $A_1 \cup \ldots \cup A_k = V$.

[4]A graph with no edges is called null graph.

1.2 Perfect Graphs

The notion of a *PERFECT GRAPH* was introduced by Claude Berge in the early part of the 60's. We know that for any graph G the minimum number of colors needed to properly color the vertices of G is atleast the size of the largest complete subgraph of G; the smallest possible clique cover of G is atleast the size of the maximum independent set of G. More formally,

$$\omega(G) \leq \chi(G) \tag{1.1}$$

$$\alpha(G) \leq \kappa(G) \tag{1.2}$$

A graph G is called perfect if it satisfies the following properties:

$$\forall A \subseteq V : \omega(G([A])) = \chi(G([A])) \tag{1.3}$$

$$\forall A \subseteq V : \alpha(G([A])) = \kappa(G([A])) \tag{1.4}$$

(Note: *As* $\alpha(G) = \omega(\overline{G})$ *and* $\kappa(G) = \chi(\overline{G})$, *if* G *is a perfect graph then so is* $\overline{G}$.).

Claude Berge conjectured that Equation 1.3 and Equation 1.4 are equivalent. Interestingly, Lovász [36] proved the conjecture and gave the following equivalent characterization [35]:

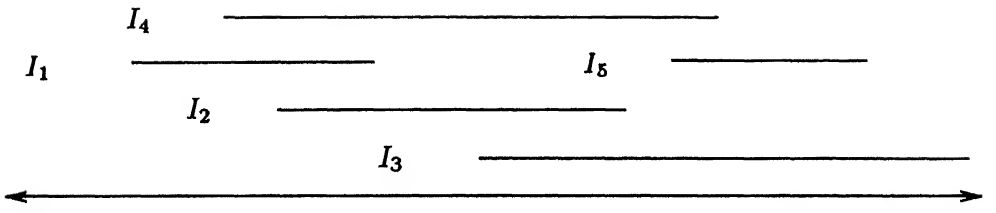
$$forall A \subseteq V : \omega(G([A]))\alpha(G([A])) \geq \mid A \mid \tag{1.5}$$

It was customary to call a graph α-perfect if it satisfies Equation 1.4; χ-perfect if it satisfies Equation 1.3. There is no simple test for recognizing perfect graphs(see Appendix A, but many different classes of perfect graphs are known. Some of them are described here with their applications.

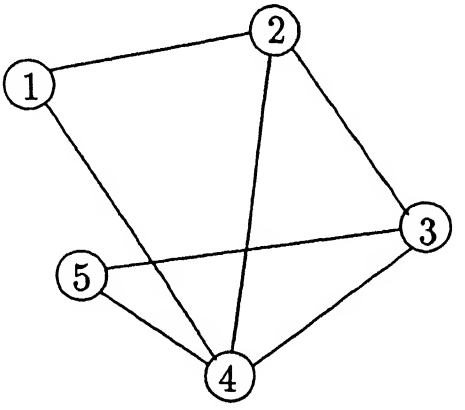
1.2.1 Interval Graphs

DEFINITION: We associate a graph G=(V,E) with the family of intervals on a real line $\mathcal{I}=\{I_i=[a_i,b_i] | a_i \leq b_i, 1 \leq i \leq n\}$, such that there is a vertex v_j for each interval I_j. Two vertices v_i and v_j are adjacent in G iff the corresponding intervals I_i and I_j overlap.

Figure 1.1(b) is an interval graph for the intervals in Figure 1.1(a).



(a) Intervals on a real line



(b) Interval graph corresponding to the intervals in (a)

Figure 1.1: Interval graph

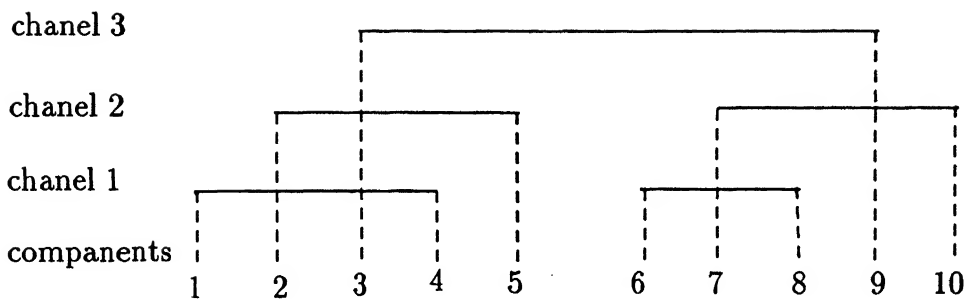


Figure 1.2: Channel routing

We next look at some applications of interval graphs.

Application 1 (Scheduling): Consider a set of courses being offered by a university. Let the courses be $c_1, c_2, \ldots, c_k$. Let t_i be the time interval for the course c_i. University wants to assign courses to classrooms so that no two courses are conducted in the same room at the same time. Construct a graph $G(V, E)$ where V is the set of courses and an edge (c_i, c_j) is in E if course c_i's time interval t_i and course c_j's time interval t_j overlap. The graph constructed so is an interval graph as the time intervals are on real line. Obviously the problem can be solved by coloring the vertices of the graph.

Application 2 (Channel routing in VLSI): Components of an electrical network are laid out in a straight line. Certain pairs of components are to be connected using only two vertical runs and one horizontal run of wires see Figure 1.2. The horizontal and vertical runs are physically located in different layers. Each horizontal wire lies in a channel; No channel can carry more than one wire at a time. The problem here is to use minimum number of channels. Consider an interval graph where each interval corresponds to a horizontal run of wire that connects a pair of electrical components. Obviously the optimal coloring of the interval graph gives the channel assignment using minimum number of channels.

Interval graphs have also found applications in archaeology, biology, psychology, management, etc. (see [25] for more details).

1.2.2 Permutation Graphs

DEFINITION: Let $\pi = [\pi_1, \pi_2, \ldots, \pi_n]$ be a permutation of length n then π_i^{-1} is the position in the permutation where the number i can be found. Permutation graph of π,

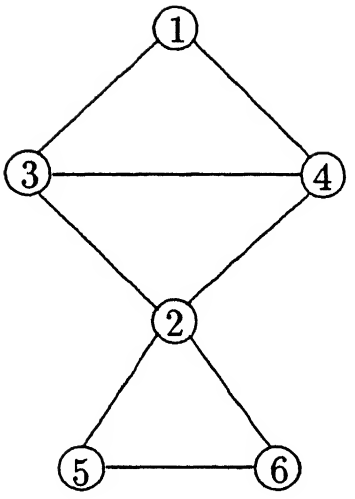


Figure 1.3: Permutation graph of $G([4,3,1,6,5,2])$

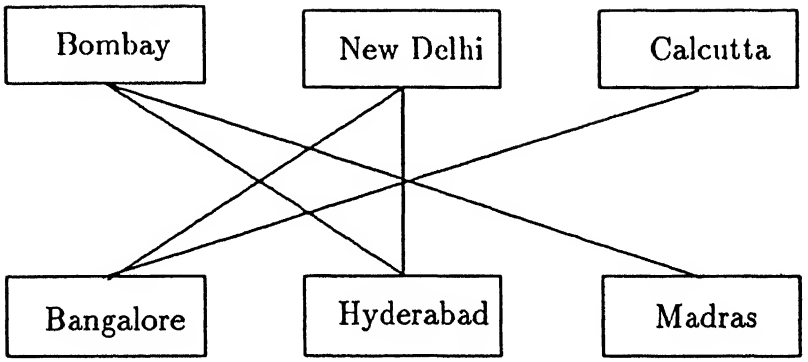
$G[\pi] = G(V,E)$ has $V = \{1,2,\cdots,n\}$ as the set of vertices and $E = \{(i,j)|(i-j)(\pi_i^{-1} - \pi_j^{-1}) < 0\}$ as the set of edges. Informally, each π_i is connected with π_{i+k} if $\pi_i > \pi_{i+k}$, for $1 \leq k \leq n-i$. An undirected graph G is called Permutation Graph if there exists a permutation π such that G is isomorphic to $G[\pi]$.

EXAMPLE For the permutation $\pi = [4,3,1,6,5,2]$ $\pi_1 = 4, \pi_2 = 3, \ldots \pi_6 = 2$ and $\pi_1^{-1} = 3, \pi_2^{-1} = 6, \ldots \pi_6^{-1} = 4$. The permutation graph $G([4,3,1,6,5,2])$ will have vertex set $V = \{1,2,3,4,5,6\}$ and edge set $E = \{(1,3),(1,4),(2,3),(2,4),(2,5)(2,6),(3,4),(5,6)\}$ (see Figure 1.3).

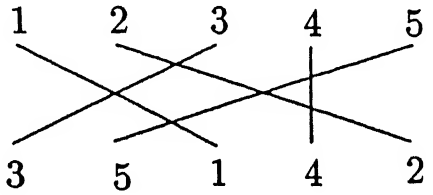
Some applications of permutation graphs are next considered.

Application 1 (Airline): Let A and B be two sets of cities. Suppose there are air line routes connecting various cities in A with various cities in B, all scheduled are to be utilized at the same time. The problem is to assign altitudes to each flight path so that intersecting routes will be at different altitudes, thereby ensuring no mid air collision. Each route will be represented by a node in the graph $G(V,E)$. Two nodes in the graph are connected if and only if the corresponding routes intersect. Coloring the nodes of the graph will solve our problem. It is clear (see Figure 1.4) that the graph so constructed will be a permutation graph.

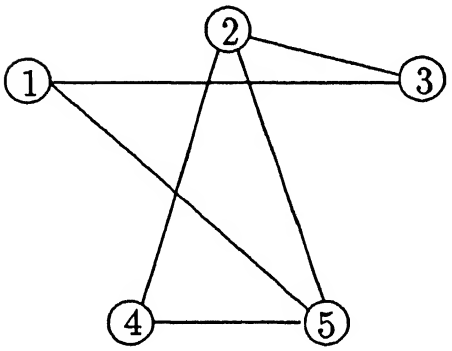
Application 2 (Multiprogramming): Let $\mathcal{I}=\{I_i=[a_i,b_i]|a_i \leq b_i, 1 \leq i \leq k\}$, be a collection of intervals on a line. Let $|\ I_i\ |$ be the length of the interval I_i, i.e. $b_i - a_i$. Assume that



(a) Airline Map



(b) π-diagram



(c) Permutation Graph

Figure 1.4: Constructing permutation graph from airline routes

the intervals (they may overlap) have been sorted such that $a_1 \leq a_2 \leq \cdots \leq a_k$. Let c_i be the cost associated with shifting the interval I_i (Note: The cost is not related to the distance it is shifted but may depend on the length of the interval). The problem asks for the minimum cost shifting of the intervals such that the following conditions are preserved:

- The order of the intervals is preserved.
- No overlap remains, i.e. $a_{i+1} \geq b_i$, for $1 \leq i \leq k-1$.

The intervals correspond to the memory requirements of k programs at a certain time in a multiprogramming environment; length of an interval corresponds to its cost.

The above problem can be modeled as a graph. Each interval represents a vertex of the graph. There is a directed edge from interval I_i to interval I_j if and only if following conditions are satisfied:

- $i < j$
- $\sum_{l=i}^{j} \mid I_l \mid \leq b_j - a_i$

In other words two intervals are related in the graph iff intervals between them can be shifted in such a way that none of these intersect. Find the maximum weighted directed path (fix the intervals on this path and shift the others). It can be shown that the graph is transitive [22]. In other words we have to find the maximum weighted clique of the undirected version of the above graph. Moreover the undirected version is a permutation graph.

1.2.3 Cographs

DEFINITION: A *cograph* is defined as follows:

a) Single vertex graph is a cograph

b) If $G_1, G_2, \ldots, G_n$ are cographs then their union $G_1 \cup G_2 \cup \ldots \cup G_n$ is a cograph

c) If G is a cograph then its complement $\overline{G}$ is also a cograph

APPLICATIONS: Many graph problems are NP-Complete on general graphs. Heuristics can be used to solve these problems on general graphs. If the given input is nearly a special graph, but not exactly a special graph then the problem is solved with the help of

the algorithm for special graphs if the algorithm on special graphs is robust enough to allow adaptation to these near misses.

Examination Scheduling: In this problem each course is represented by a node and node i, node j are connected by an edge iff some students take courses corresponding to the nodes i and j. Examination for the courses in the same color class can be conducted concurrently. In practice the graph given to this problem is close to a cograph [18].

1.3 Overview of the Thesis

Chapter 2 describes the parallel computational model used in the thesis— the well-known *Parallel Random Access Machine* (PRAM) model. All algorithms in this thesis assume this model. Problem of finding k-connected components of an interval graph is described in Chapter 3. In Chapter 4 an optimal parallel algorithm for prefix computation on trees is described. The algorithm takes $O(\log n)$ time with $\frac{n}{\log n}$ processors on an *Exclusive Read and Exclusive Write* (EREW) PRAM model. In Chapter 5 the algorithm given in [37], for computing a longest common sub-sequence is improved with the help of the prefix computation on trees 4. In the same chapter lower bounds for finding the size of the maximum independent set of a permutation graph are considered. As permutation graph is a perfect graph, the same lower bounds apply to chromatic number as well. Optimal parallel algorithms for permutation representation and minimum coloring of cographs are discussed in Chapter 6. Finally, in chapter 7 some conclusions are offered and scope for future work is discussed.

Chapter 2

Parallel Computational Model

Parallel Random Access Machine (PRAM) model is perhaps the most popular model of parallel computers for algorithmic design. It neglects any hardware constraints and is ideal for studying inherent parallelism present in a problem. It gives absolute freedom to algorithm designer in presentation of parallel algorithms. In any realization of PRAM there will be a link between each processor and each memory location. This is however not realizable with the present day architectures. There are methods of simulating (like sorting on address and processor index) such an idealized computer on more reasonable parallel computers (like fixed networks of processors with number of linkages from any processor being bounded). This simulation costs only polylogarithmic time on most models [5].

PRAM is a shared memory model. Many processors work synchronously and communicate through the common random access memory. Each processor is a uniform cost Random Access Machine (RAM), with usual operations and instructions (see e.g., [3] for details of RAM model).

The processors are indexed by consecutive integers usually from the number 0 (i.e. 0–$(n-1)$ if there are n processors). See the Figure 2.1 All processors execute the same instruction but (possibly) on different data. Hence it is a single instruction multiple data (SIMD) stream model.

In one step each processor either reads or writes into a memory location. There are four models in PRAM family depending on rules for the simultaneous access of the same memory location.

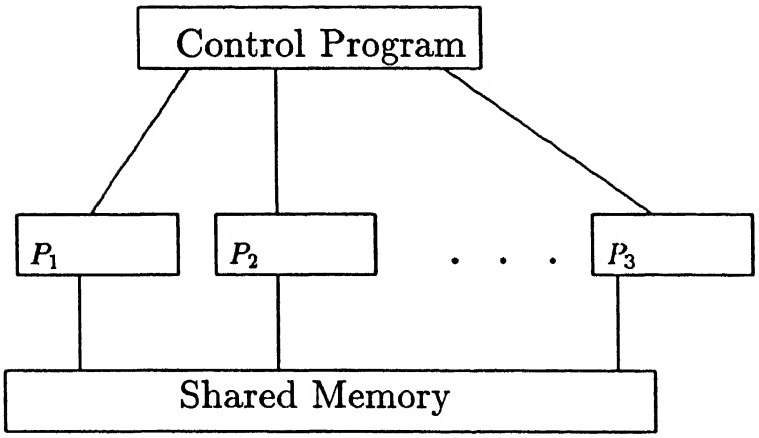


Figure 2.1: PRAM model

EREW PRAM: EREW is an acronym for Exclusive Read and Exclusive Write. In this model no two processors can attempt to read the same memory location at the same time; nor two processors can attempt to write into the same memory location at the same time. This is the weakest model among all PRAM models.

CREW PRAM: This stands for Concurrent Read and Exclusive Write. In this model two or more processors can read the same memory location at same time. But still no two processors can try to write into the same memory location simultaneously.

CRCW PRAM: In this model, simultaneous concurrent reads and concurrent writes are permitted. There are many variants of CRCW PRAM based on the write conflict resolution rule. Some of these are:

PRIORITY: If the highest numbered (highest priority) processor succeeds in writing then the model is called PRIORITY concurrent write PRAM.

ARBITRARY: If it is not known which processor succeeds, but some single processor is guaranteed to succeed then the model is called ARBITRARY PRAM model.

TOLERANT: The model is called TOLERANT if no processor succeeds in case of multiple writes and contents of the memory location are guaranteed to remain unchanged.

COMMON: In this model every processor is required to write the same value in case of multiple writes into the same memory location. This value appears that location.

See [23, 43] for simulation of one model on the other.

Chapter 3

Connectivity Problems for Interval Graphs

We associate a graph G=(V,E) with the family of intervals $\mathcal{I}=\{I_i=[a_i,b_i]|a_i \leq b_i, 1 \leq i \leq n\}$, such that there is a vertex v_j for each interval I_j. Two vertices v_i and v_j are adjacent in G iff the corresponding intervals I_i and I_j overlap.

Interval graphs have found applications in archaeology, biology, psychology, management, VLSI design, scheduling, etc. (see [25] for more details).

Efficient parallel algorithms have been devised for various problems for interval graphs [11, 42, 47]. In this thesis an $O(\log\log n)$ time algorithm with $\frac{n}{\log\log n}$ processors for finding biconnected components is obtained on a COMMON CRCW PRAM, assuming that the intervals are given in sorted order. This algorithm can also be implemented on a TOLERANT CRCW PRAM with same resource bounds. We say $I_i > I_j$ iff $a_i > a_j$. The intervals are sorted in increasing order if $I_1 < I_2 < \cdots < I_n$. Further, if the end points are integers then the time to find biconnected and k-connected components can be reduced to[1] $O(\log^* n)$ and $O(k \log^* n)$, respectively with $\frac{n}{\log^* n}$ processors on a Priority write PRAM.

Earlier Liang, Dhall and Lakshmivarahan [34] have obtained $O(\log n)$ time algorithm for finding biconnected components with n processors on a CREW PRAM. Sprague and Kulkarni [45] have given $O(\log n)$ time optimal (on sorted intervals) algorithm for finding biconnected components on an EREW PRAM. Das, Calvin and Chen [19] have obtained

[1] $\log^* n = \min\{i | \log^{(i)} n \leq 1\}$

$O(\log n)$ time algorithms for finding all bridges and articulation points with n processors on an EREW PRAM.

We generalize the notion of "prefix-maxima" to "k-th-prefix-maxima". All algorithms are based on the observation that k-connected components can be easily identified if "k-th-prefix-maxima" are known. We believe that the notion of "k-th-prefix-maxima" will be useful in other applications also.

3.1 K-th Prefix Maxima

Given a collection of intervals $\mathcal{I}=\{[a_i,b_i]|1 \leq i \leq n\}$. We say a_i and b_i are the *left end point* and the *right end point* of i^{th} interval respectively. We will say two intervals are adjacent, if the corresponding vertices in associated graph are adjacent.

Right Interval: I_r is called the right interval of set $\mathcal{S}$ if $b_r = \min\{b_i|i \in \mathcal{S}\}$.

Left Interval: I_l is called the left interval of set $\mathcal{S}$ if $a_l = \max\{a_i|i \in \mathcal{S}\}$.

We define *k-th-prefix-maxima*(I_i) to be the k^{th} largest element in the set $S_i = \{b_j|a_j \leq a_i\}$. If k-th-prefix-maxima$I(I_i)$ is b_m then we say interval I_m *contributed* to k-th-prefix-maxima(I_i). We say interval I_i is *critical* in the j^{th} iteration iff j-th-prefix-maxima(I_i) is greater than j-th-prefix-maxima(I_{i-1}). We say that a set of intervals $\mathcal{I}=\{I_i\}$ *spans* [a,b], if for every point x, $a \leq x \leq b$, there is an interval I_j, such that $a_j \leq x \leq b_j$

The k-th-prefix-maxima (for $k \geq 2$) of each interval can be found as follows.

Step 1: Let C[j]:=b_j for $1 \leq j \leq n$.

Step 2: Find prefix maxima of the array C. Let array D be the prefix maxima of array C.

Step 3: For each critical interval I_j (*i.e.* D[j]$\neq$D[j-1]) do C[j]:=D[j-1].

Step 4: Repeat step(2) followed by step(3), $k-2$ more times.

Step 5: Find prefix maxima of array C.

The last prefix maxima will give us the k-th-prefixmaxima.

Prefix maxima can be found optimally (with a processor-time product of O(n)) in $O(\log n)$ time on an EREW PRAM, and in $O(\log\log n)$ time on a COMMON CRCW PRAM [9, 8]. If the items are integers then prefix maxima can be found in optimal $O(\log^* n)$ time

on a Priority write PRAM [8, 9].

The usual "1-st-prefix-maxima" will be called prefix maxima.

3.2 Connected Components

We compute connected components using 2-nd-prefix-maxima. Note that if for any interval 2-nd-prefix-maxima(I_i) < a_i then the 1-st-prefix-maxima(I_i) will be b_i itself. Hence there is no interval spanning the real line from 2-nd-prefix-maxima(I_i) to a_i. Or, I_i starts a new connected component.

All intervals which start a new component can be identified (2-nd-prefix-maxima(I_i) < a_i) and marked. Let A[i] be ∞ if interval I_i is not marked, and let A[i]=i, if interval I_i is marked. Then find the left nearest smaller of array A. [10, 51]. Thus each interval is identified by the smallest interval in that component. The above procedure takes $O(\log\log n)$ optimal time on a COMMON CRCW PRAM and $O(\log n)$ optimal time on an EREW PRAM.

Theorem 3.1 *Connected components of interval graph can be found in $O(\log\log n)$ optimal time on a COMMON CRCW PRAM.* ∎

We can do better on a PRIORITY write PRAM, if the end points are integers. We proceed as follows

> A flag is set to 1 if the interval is marked and it is reset to 0 otherwise. We "connect" all 1's by using ordered chaining algorithm of [41] in $O(\alpha(n))$ optimal time on a COMMON CRCW PRAM. Each 0 can find the index of the immediately previous 1 by using the algorithm of [9] for the nearest one complement problem. This operation also takes $O(\alpha(n))$ optimal time on a COMMON write PRAM.

Theorem 3.2 *Connected components of interval graph can be found in $O(\log^* n)$ time with $\frac{n}{\log^* n}$ processors on Priority write PRAM if the end points are integers.* ∎

Note that if all end points are given in sorted order, then we can replace each end point by an integer.

3.3 Biconnected Components

Biconnected components of a connected graph G are found by eliminating all articulation points of G and finding connected components of the resulting graph G'. Articulation points are found by using 2-nd-prefix-maxima algorithm.

If 2-nd-prefixmaxima(I_{i-1}) $< a_i$ then the interval which contributed to prefix maxima is an articulation point, provided 2-nd-prefix-maxima(I_{i-1}) > 0. This can be seen as follows. Let us assume that I_f contributed to 1-st-prefix-maxima(I_{i-1}) and I_s to the 2-nd-prefix-maxima(I_{i-1}). Thus 2-nd-prefix-maxima(I_{i-1}) $\equiv b_s < a_i$. Moreover, as G is connected, $b_f > a_i$. Thus, the real line from b_s to a_i is spanned by only one interval, I_f. Hence, I_f is an articulation point.

We identify and then remove all the articulation points of G, and find the connected components of the resulting graph G'. In more detail, we proceed as follows.

If interval I_i is an articulation point for components $C_{j1}, C_{j2}, C_{j3}, \cdots, C_{jm}$ then every component's first interval (smallest numbered interval in that component) except the first component C_{j1} detects the articulation point (using the prefix maxima routine). Then, all detected articulation points are removed, and connected components are found using the algorithm of Section 3.2. We again, "link" all first intervals of each connected component as in Section 3.2; each component corresponds to a bi-connected component. We next describe the method for recording each articulation point in all components (to which it is adjacent). Recall, that every component's first interval, except the first component C_{j1} knows the articulation point. We still have to inform the first component.

Observe that each bi-connected component can be adjacent to at most two articulation points, one adjacent to first interval of the component, and other to the interval which ends last. We will use an array A of length $2n$ to record articulation points. If I_j is first interval of a bi-connected component, then the articulation points will be available in locations A[2j-1] and A[2j].

- If interval I_i detects articulation point I_j then record I_j at location $2i$ in the array A.
- Let I_k be the first interval of a component just before "component" I_i then record articulation point I_j at location $2k + 1$ in the array A.

Thus, each interval knows the bi-connected component in which it lies (first interval

of the bi-connected component) and each first interval I_j has pointers to at most both articulation points. This, completely characterizes the biconnected components.

Theorem 3.3 *Biconnected components of interval graph can be found in* $O(\log\log n)$ *time optimally on a COMMON CRCW PRAM model.* ∎

Theorem 3.4 *Biconnected components of interval graph can be found in* $O(\log^* n)$ *time optimally on Priority write PRAM if the end points are integers.* ∎

3.4 k-Connected Components

Given a positive integer k, a graph G=(V,E) with at least $k+1$ vertices is called *k-vertex connected* if deletion of any $k-1$ vertices leaves the graph connected. Testing k-connectivity of a general graph [13] with n vertices and m edges takes $O(k^2 \log n)$ time using $\frac{kn(n+m)\alpha(n)}{\log n}$ processors on CRCW PRAM.

For simplicity, we assume that the graph G is $k-1$ connected. We say a set of vertices form a *disconnecting set* if removal of all vertices in the set disconnects the graph. As the graph is $k-1$ connected, it follows that every pair of vertices in a disconnecting set are adjacent; and in each component adjacent to the disconnecting set there is at least a vertex which is adjacent to all vertices of the disconnecting set. Further, each k-connected component can be adjacent to at most two different disconnecting sets. We will find k-connected components by eliminating all maximal disconnecting sets of size $k-1$.

Observe that if *k-th-prefix-maxima*$(I_{i-1}) < a_i$ then all those intervals which contributed to 1-st-prefix-maxima(I_{i-1}), 2-nd-prefix-maxima$(I_{i-1}), \cdots,$ $k-1$-th-prefix-maxima(I_{i-1}) will form a disconnecting set of size $k-1$.

Lemma 3.1 *For each disconnecting set of size* $k-1$ *there exists an interval* j *such that k-th-prefix-maxima*$(I_{j-1}) < a_j$.

Proof: Let $\mathcal{D}$ be a disconnecting set of size $k-1$. Let $\mathcal{I}'$ be the set of intervals obtained by removing all the intervals of the disconnecting set $\mathcal{D}$. Let C_1 and C_2 be the first two components of the resulting graph. Let I_r be the right interval of C_1 and let I_j be the first interval of C_2. As C_1 and C_2 are in different components in $\mathcal{I}'$, prefix-maxima$'(a_r) < a_j$.

As there $k-1$ intervals in disconnecting set, it follows that k-th-prefix-maxima(a_r) $< a_j$, and for any $I_{j'} \in \mathcal{D}$ k-th-prefix-maxima($a_{j'}$) $< a_j$. Thus, the lemma follows. ∎

Thus, all k-connected components can be found by first removing all disconnecting sets of size $k-1$ and then use connected components algorithm on the resulting graph; connected components of the resulting graph will correspond to k-connected components of the original graph. Vertices of disconnecting sets can be made known to the components adjacent to them by a method similar to that described in Section 3.3. However, as there are $k-1$ vertices in disconnecting set, we will have to use $k-1$ arrays $A_1, \cdots A_k$, each of length 2n.

Theorem 3.5 *k-connected components of interval graph can be found in* $O(k \log\log n)$ *time with* $(\frac{n}{\log\log n})$ *processors on a COMMON CRCW PRAM model.* ∎

Theorem 3.6 *k-connected components of interval graph can be found in* $O(k \log^* n)$ *time with* $(\frac{n}{\log^* n})$ *processors on Priority write PRAM if the end points are integers.* ∎

Chapter 4

Prefix Computation on Trees

Prefix computation on lists and arrays is a well known problem; it is a very useful routine in graph theoretical problems (see e.g., [33]). Generalization of this problem on trees is:

> Given a tree T and an associative binary operation $\odot$, compute for each node v: $v_1 \odot v_2 \odot \cdots \odot v_k$; here v_1 is parent(v), v_k is the root of T and v_{i+1} is parent(v_i), for $1 \leq i \leq k-1$.

Prefix computation problem for linked list can be reduced to the prefix computation problem for arrays using list ranking. List ranking can be solved in O($\log n$) time with $\frac{n}{\log n}$ processors on an Exclusive Read Exclusive Write (EREW) PRAM [16, 6]. Prefix computation on an array can be done optimally in $O(\log n)$ time (see e.g., [33]) on an EREW PRAM.

Remark : Another generalization of the prefix computation problem is the so called "General prefix computation" problem [46]: Let X, Y be two given arrays and an associative binary operation $\odot$ be defined on elements of X; elements of Y can be compared by a linear order $<$. Then for each index i we are required to compute $Z_i = X_{j_1} \odot X_{j_2} \odot \cdots \odot X_{j_k}$ where $j_1 < j_2 < \cdots < j_k < i$ and $\{j_1, j_2, \ldots, j_m, \ldots j_k\}$ is the set of indices for which $Y_{j_m} < Y_i$ for $1 \leq m \leq k$. In other words for each X_i we "add" only those X_j for which a) $j \leq i$ b) $Y_j \leq Y_i$. Sorting can be reduced to general prefix computation problem (Y consists of items to be sorted and X consists of n 1's). An O($\log n$) time, n processor parallel algorithm for this problem has been obtained in [46].

Remark : An $O(\log n)$ time parallel algorithm for Prefix Computation on Trees with n processors can be obtained by using the "General Prefix Computation" algorithm of [46]. Traverse the tree in pre-order and pack the given items into array X according to their pre-order number; array Y contains the *rep-order* number of nodes; "rep-order" traversal can be described as follows: visit a node, then visit the children in *right to left* order. Rep-order number can be found by first reversing the order of children of each node and then traversing the tree in pre-order.

4.1 Algorithm

The given tree T is first converted into a regular binary tree B (each node has two or no children) using the method of [32]:
If a node v has k children, $v_1, v_2, \ldots, v_k$ then introduce k additional nodes $v'_1, v'_2, \ldots, v'_k$ such that

- v'_1 is a right child of v.
- v'_j is a right child of v'_{j-1}, for $2 \leq j \leq k$
- v_1 is a left child of v.
- v_j is a left child of v'_{j-1}, for $2 \leq j \leq k$

Note that the ancestors of a node in T are also ancestors of the corresponding node in B. But the set of ancestors of a node in B may include some new nodes introduced during the conversion. If the identity element of the operator $\odot$, is put in the newly introduced nodes, the problem of prefix computation on T can be solved once the problem is solved for B. Further, note that the number of nodes introduced is always less than n. Thus, without loss of generality, in the rest of the chapter, we may assume that the given tree is a binary tree.

The algorithm successively modifies the given tree T into a sequence of "simpler" trees $T_1, T_2, \ldots, T_k$ such that
a) the final tree T_k is simple enough to obtain prefix computation in $O(1)$ time
b) the prefix computation on T_i can be done in $O(1)$ time using the "knowledge" of prefix computation on T_{i+1} and
c) k is small, i.e., $k = O(\log n)$.

All these conditions are satisfied for the tree contraction sequence generated by algorithm of [1]. The tree contraction sequence is defined as follows[1]:

DEFINITION : Let B be a binary tree with vertex set $V(B)$. A sequence of trees $B_1, B_2, \cdots, B_k$ is said to be a tree contraction sequence of length k for B if,

1. $B_1 = B$
2. $V(B_i) \subseteq V(B_{i-1})$
3. $| V(B_k) | \leq 3$
4. If $v \in V(B_{i-1}) - V(B_i)$ then either
   i) v is leaf of B_{i-1} or
   ii) v has exactly one child, x, in B_{i-1}, $x \in V(B_i)$, and the parent of v in B_{i-1} is the parent of x in B_i.

We use the basic algorithm of [1] for tree contraction. How-ever some extra bookkeeping information is maintained in routines — prune[ ] and bypass[ ].

Algorithm: We assume that the field "Value[v]" stores the value initially at node v, or the identity element of "$\odot$". Number the leaf nodes from left to right s.t., for left-most leaf v_l, Number[v_l]=0 and for right-most node v_r, Number[v_r]=$n-1$. The algorithm is as follows:

```
for i=1 to ⌈log n⌉ − 1 do serially
{
   for each leaf v pardo
      { w:=parent(v);
        if Number(v) is odd and w ≠ root
                    if v is a left child of w then j:=0; prune(v); j:=1; bypass(w);
                    if v is a right child of w then j:=2; prune(v); j:=3; bypass(w);
        else Number(v):=Number(v)/2;
      }
}
```

prune(v)

{

Index[v]:= $4i + j$;

Last_Parent[v]:= Parent[v]

Remove the node v as in the usual "Prune" routine

}

bypass(v)

{

Index[v]:= $4i + j$;

Last_Parent[v]:= Parent[v]

If x is the only child of v then

parent(x) := parent(v)

value(x) := value(x)$\odot$ value(v)

Remove the node v as in the usual "Bypass" routine

}

Note : If Index[v] is "l", then the node v was in trees $B_1, \ldots, B_l$, but was not in tree B_{l+1}.

Lemma 4.1 *Let B_i be the i^{th} tree in the tree contraction sequence, w be the parent of node v in B_i and $v_1, v_2, \ldots, v_j$ be the proper ancestors of v and proper descendants of w (i.e. $v_1, v_2, \ldots, v_j$ are the nodes on the path from v to w) in B. Before computation of B_{i+1}, the value(v) will be Init_value(v) $\odot$ Init_value(v_1) $\odot \cdots \odot$ Init_value(v_j) where Init_value represents the initial value given to the nodes.*

Proof: : The lemma is proved by induction. The lemma is trivially true for B_1. Assume that it is true for B_i.

Assume that node u is bypassed in B_i to get B_{i+1}. Let v be the only child of u and w be parent of u in B_i. By induction hypothesis

```
prune(v)
{
  Index[v]:= 4i + j;
  Last_Parent[v]:= Parent[v]
  Remove the node v as in the usual "Prune" routine
}

bypass(v)
{
  Index[v]:= 4i + j;
  Last_Parent[v]:= Parent[v]
  If x is the only child of v then
       parent(x) := parent(v)
       value(x) := value(x)⊙ value(v)
  Remove the node v as in the usual "Bypass" routine
}
```

Note : If Index[v] is "l", then the node v was in trees $B_1, \ldots, B_l$, but was not in tree B_{l+1}.

Lemma 4.1 *Let B_i be the i^{th} tree in the tree contraction sequence, w be the parent of node v in B_i and $v_1, v_2, \ldots, v_j$ be the proper ancestors of v and proper descendants of w (i.e. $v_1, v_2, \ldots, v_j$ are the nodes on the path from v to w) in B. Before computation of B_{i+1}, the value(v) will be Init_value(v) $\odot$ Init_value(v_1) $\odot \cdots \odot$ Init_value(v_j) where Init_value represents the initial value given to the nodes.*

Proof: : The lemma is proved by induction. The lemma is trivially true for B_1. Assume that it is true for B_i.

Assume that node u is bypassed in B_i to get B_{i+1}. Let v be the only child of u and w be parent of u in B_i. By induction hypothesis

value(v)= Init_value(v) $\odot$ Init_value(v_1) $\odot \cdots \odot$ Init_value(v_m) and
value(u)= Init_value(u) $\odot$ Init_value(u_1) $\odot \cdots \odot$ Init_value(u_n)
where nodes $v_1, v_2, \ldots, v_m$ are on the path from v to u and $u_1, v_2, \ldots, u_n$ are on the path from u to w. After bypassing node u (see bypass routine), value(v) will be
value(v)= Init_value(v) $\odot$ Init_value(v_1) $\odot \cdots \odot$ Init_value(v_m) $\odot$ Init_value(u) $\odot$ Init_value(u_1) $\odot \cdots \odot$ Init_value(u_n).
All these nodes are on the path from v to w in B (w is parent of v in B_{i+1}). ∎

Corollary 4.1 *Let Index[v] be i and w=Last_parent[v], be the parent(v) in B_i. When the algorithm terminates the value(v) will be*
Init_value(v) $\odot$ Init_value(v_1) $\odot \cdots \odot$ Init_value(v_j)
where $v_1, v_2, \ldots, v_j$ are nodes on the path from v to w in B. ∎

The last tree in the sequence, B_k has three nodes, r, the root of tree, c_1, left child of r and c_2, right child of r. By Corollary 4.1, final value of c_1 and c_2 can be calculated as follows:
value(c_1) := value(c_1)$\odot$ value(r)
value(c_2) := value(c_2)$\odot$ value(r)

Assume that the final value of each node in B_{i+1} has been calculated. The value of nodes in B_i but not in B_{i+1} can be calculated as follows: Let v be a node in B_i but not in B_{i+1}; i.e., Index[v]=i. Let w=Last_Parent[v], be the parent of v in B_i then value(w) will be Init_value(w) $\odot$ Init_value(w_1) $\odot \cdots \odot$ Init_value(w_j) where $w_1, w_2, \ldots, w_r$ are proper ancestors of w in B and value(v) will be
Init_value(v) $\odot$ Init_value(v_1) $\odot \cdots \odot$ Init_value(v_j) where $v_1, v_2, \ldots, v_j$ are nodes on the path from v to w in B. Final value(v) will be: value(v):=value(v) $\odot$ value(w).

As the maximum value of "Index[ ]" is $4 \log n$, the algorithm can easily be implemented in O($\log n$) time with n processors. Using sorting algorithm of Cole and Vishkin [15], processor-allocation problem can be solved and the number of processors reduced to $\frac{n}{\log n}$ without any increase in time. In more details, we proceed as follows: As "Index[ ]" is in the range $1, \ldots, 4 \log n$, the nodes can be sorted on the key Index[v], in $O(\log n)$ time with $\frac{n}{\log n}$ processors [15]. Calculate each node's RANK as, RANK[v]:= POSITION(v)+$m\frac{n}{\log n}$ where m=Index[v] and POSITION(v) is the position of node v in the sorted array. We process nodes in $2 \log n$ rounds— In j^{th} round all vertices whose RANK lies between ($2 \log n$ –

$j-1)(\frac{n}{\log n})$ and $(2\log n - j)(\frac{n}{\log n})$ are processed. This ensures that no two vertices with different Index[ ] (or in different trees) are processed in the same round.

Let C_i be the number of nodes in B_i but not in B_{i+1}. Time taken in i*th* round is $\max(\frac{C_i \log n}{n}, 1)$; thus time taken by algorithm is atmost
$\sum_{i=1}^{4\log n} max(\frac{C_i \log n}{n}, 1) \leq \sum_{i=1}^{4\log n}(\frac{C_i \log n}{n} + 1) = \frac{\log n}{n}\sum_{i=1}^{4\log n} C_i + 4\log n = \log n + 4\log n = 5\log n = O(\log n)$
Thus, the following theorem follows:

Theorem 4.1 *Prefix computation on trees can be done in* $O(\log n)$ *time using* $\frac{n}{\log n}$ *processors on an EREW PRAM.* ∎

4.2 Applications

Applications of prefix computation on trees are discussed in Chapter 5 and Chapter 6. This algorithm has applications for computing a largest increasing sub-sequence which in turn can be used to compute maximum independent set of a permutation graph. Previous algorithm for computing largest increasing sub-sequence[37] takes $O(\log^3 n)$ time with n processors on a CREW PRAM. The processor-time product can be reduced from $O(n \log^3 n)$ to $O(n \log^{2.5} n)$ on a CREW PRAM; see Chapter 5 for details. Prefix computation on trees is also used for coloring and permutation representation of cographs; see Chapter 6.

Chapter 5

Maximum Independent Set of a Permutation Graph

5.1 Introduction

Let $\pi = [\pi_1, \pi_2, \ldots, \pi_n]$ be a permutation of length n then π_i^{-1} is the position in the permutation where the number i can be found. Permutation graph of π, $G[\pi] = G(V, E)$ has $V = \{1, 2, \cdots, n\}$ as the set of vertices and $E = \{(i,j) | (i-j)(\pi_i^{-1} - \pi_j^{-1}) < 0\}$ as the set of edges.

Informally, each π_i is connected with π_{i+k} if $\pi_i > \pi_{i+k}$, for $1 \leq k \leq n-i$. An undirected graph G is called Permutation Graph if there exists a permutation π such that G is isomorphic to $G[\pi]$.

EXAMPLE For the permutation $\pi = [4,3,1,6,5,2]$ $\pi_1 = 4, \pi_2 = 3, \ldots \pi_6 = 2$ and $\pi_1^{-1} = 3, \pi_2^{-1} = 6, \ldots \pi_6^{-1} = 4$. The permutation graph $G([4,3,1,6,5,2])$ will have vertex set $V = \{1,2,3,4,5,6\}$ and edge set $E = \{(1,3),(1,4),(2,3),(2,4),(2,5)(2,6),(3,4),(5,6)\}$. ∎

Permutation graphs were introduced by Even and Pnueli in 1972 [21]; for applications of permutation graphs see e.g.[25, 22]. Spinrad gave $O(n^2)$ algorithm for recognizing permutation graphs [44]. Finding maximum independent set of a general graph is NP-complete [24], but maximum independent set of a permutation graph can be found in polynomial time [31, 52].

Next section describes the algorithm for finding maximum independent set of a permu-

tation graph. Lower bounds for finding maximum independent set of a permutation graph, given its permutation representation, are presented in Section 5.3.

5.2 Algorithm for Maximum Independent Set

String S is said to be a sub-sequence of string A if either a) S is equal to A or b) S is obtained by deleting some symbols from A. String C is a common sub-sequence of string A and string B if C is a sub-sequence of A as well of B. A string having maximum length among all common sub-sequences is called a longest common sub-sequence. The longest common sub-sequence problem can be used for computing maximum independent set of a permutation graph; for other applications of longest common sub-sequence problem see [38, 14].

5.2.1 Preliminaries

Sub-sequence $S = s_1, s_2, \cdots, s_r$ of a string A is called an increasing (decreasing) sub-sequence if $s_{i+1} > s_i$ ($s_{i+1} < s_i$) for $1 \leq i \leq r-1$. Increasing (decreasing) sub-sequence of string A which has maximum length among its increasing (decreasing) subsequences is called Longest Increasing (Decreasing) Sub-sequence. It is known that [25] increasing (decreasing) sub-sequence of a permutation corresponds to independent set (clique) of the graph represented by the permutation. Hence a Longest Increasing (Decreasing) Sub-sequence corresponds to a Maximum Cardinality Independent set (Maximum cardinality Clique) of the corresponding permutation graph.

Largest Increasing sub-sequence can be obtained by finding the longest common sub-sequence of the given string and the string $[1, 2, \cdots, n]$.

5.2.2 Algorithm

The algorithm of [37] for finding the Longest Common Sub-sequence takes $O(\log^3 n)$ with $\max(r, n)$ processors, where r is the total number of positions at which the two input strings match and n is the length of the longer string. Some of the steps in the algorithm can be implemented at lesser cost with the help of prefix computation on trees . For completeness

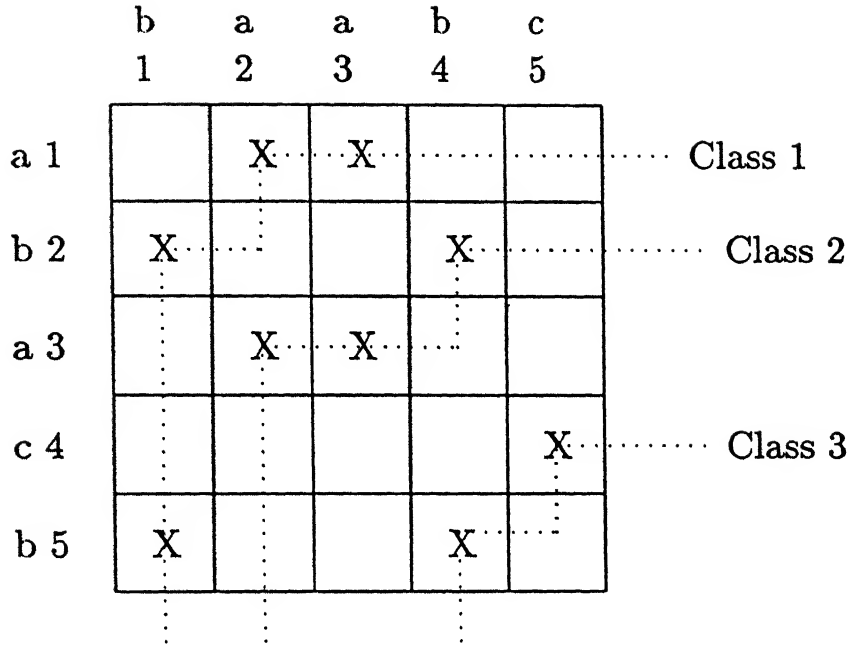


Figure 5.1: Lu's Algorithm

the algorithm of [37] is breifly described below (for details refer to [37]).

Let $A = a_1, a_2, \cdots, a_n$ and $B = b_1, b_2, \cdots, b_m$ be the two strings. Let $G[i,j]$ be the length of longest common sub-sequence of strings $a_1, \ldots, a_i$ and $b_1, \ldots, b_j$. Only matched symbols are relevant in constructing a longest common sub-sequence. $G[i,j]$ is computed only if $a_i = b_j$ for $1 \leq i \leq n; 1 \leq j \leq m$. The selected entries, where $a_i = b_j$, on the grid (see Figure 5.1) are called *points* and the value $G[i,j]$ is called *class* of point (i,j). Point $q(i_q, j_q)$ is said to be *dominated* by point $p(i_p, j_p)$ iff $i_p < i_q$ and $j_p < j_q$. Point $f(i_f, j_f)$ can be the father of point $s(i_s, j_s)$ if

1. point f dominates point s

2. class(f) is one less than the class(s).

A processor is assigned to each pair (i,j) if $a_i = b_j$. This scheduling step can be done in $O(\log m + \log n)$ time using $m + n$ processors (see [37]).

Divide the grid into two parts by a horizontal line. Solve the problem (i.e., find the class of each point in that half) for each half concurrently and merge the two solutions. After the merge the class of each point in the upper half will remain the same, but the points in

the lower half might have got their class changed. Hence, in the merge step we have to find the right class for each point in the lower half.

The merge step is carried out as follows: Project the left most point of each class in the upper half onto the horizontal dividing line. Denote the projection as joint. Locate a dummy joint at the left side of the grid on the horizontal line. Merge consists of two phases. In phase I each point in the lower half finds its father. A point in class 1 finds its father among the joints. Hence a forest is formed with joints as the roots. Roots will inform their class numbers to their descendants. Each point can decide its new class in phase I by adding its root's class to its depth in the tree. This phase is called *update with joint.*

The second phase of the merge step is *update between trees* i.e. merging the trees formed in the first phase. This phase consists of $\log t$ iterations if there are t trees. In iteration i, 2^i trees should be merged. Let us call the left 2^{i-1} trees as *left trees* and similarly the right 2^{i-1} trees as *right trees.* Each point p finds a point t in the right trees such that the point t dominates point p and point t has maximum class number among such points. Then the point p changes its class number to $\max(class(p), class(t) + 1)$. The following is a brief description:

Algorithm LCS

Begin

Divide the input grid into two parts, upper half and lower half

Recursively find the class numbers of each part independently and concurrently

Merge the two parts

End.

Procedure Merge

/ Update with Joint*/*

Begin

Find Joints from the upper half

Sort the points in the lower half lexicographically on the key (class number, column number)

Each point in lower half finds its father in the class above it;

however a point in class 1 finds its father from the Joints. Thus forming trees.

Perform propagation_1(class) on each tree; i.e. each point p gets $class(root(p)) + depth(p)$

/* *Update between trees*/*

For $\log\sigma + 1$ times do serially /* σ = number of Joints*/

Each point $p(i_p, j_p)$ finds the point $q(i_q, j_q)$ in the right trees such that

$j_p > j_q$ and q has the maximum class number among the points that dominate p.

diff(p)=class(q)-class(p)

Perform propagation_2(diff); i.e. each point gets maximum diff among its ancestors

class(p):=class(p)+diff(p)+1

if diff(p) > diff(father(p)) then father(p):=q

End

First improvement comes from the fact that we can use integer sorting instead of comparison based sorting. Integer sorting of n numbers in the range $[1 \cdots n]$ takes

1. $O(\log n\sqrt{\log n})$ with $(\frac{n}{\log n})$ processors on an EREW PRAM [4].
2. $O(\frac{\log n}{\log\log n}$ time with $n\frac{(\log\log n)^2}{\log n})$ processors on a ARBITRARY-write CRCW PRAM [12].

Operations propagation_1 and propagation_2 can be performed in $O(\log n)$ time with $(\frac{r}{\log n})$ processors on an EREW PRAM as follows:

Note that class of each point will be its level number in the tree. Class of root (Joint) can be propogated by assigning $n + 1$ to every other node and then performing prefix computation on the tree 4 with minima as the associative binary operation. The class of a node p after propagation_1 will be class(root)+class(p). Similarly, propagation_2 can again be done by prefix computation on tree with maxima as the associative binary operation.

Algorithm for prefix computation on trees assumes Adjacency List Representation. We can sort the edges using Integer Sorting algorithm [4, 12] to convert the representation to adjacency list.

Hence the following theorem,

Theorem 5.1 *The algorithm for finding a longest common sub-sequence of two given strings takes*

1. $O(\log^3 \sqrt{\log n})$ *time with* $\max(\frac{n}{\log n}, \frac{r}{\log n})$ *processors on an EREW PRAM model*
2. $O(\log^3)$ *time with* $\max(n\frac{(\log\log n)^2}{\log n}, r\frac{(\log\log n)^2}{\log n})$ *processors on an ARBITRARY-write CRCW PRAM*

where r is the total number of positions at which the two given strings match.

Corollary 5.1 *The algorithm for finding a maximum cardinality independent set or a maximum cardinality clique of a Permutation graph takes*

1. $O(\log^3 n\sqrt{\log n})$ *with* $(\frac{n}{\log n})$ *processors on an EREW PRAM.*
2. $O(\log^3 n)$ *time with* $n\frac{(\log\log n)^2}{\log n}$ *processors on an ARBITRARY-write CRCW PRAM.*

5.3 Lower bounds for the maximum independent set

Given a binary string[1] (with alphabet $\{0, 1\}$) $A = a_1, a_2, \cdots, a_n$, we construct permutation π by looking for instances of 10's (that is digit 1 immediately followed by digit 0) in A as follows:
If $a_i = 1$ and $a_{i+1} = 0$ then let $\pi_i = i + 1$ and $\pi_{i+1} = i$. In all other cases let $\pi_i = i$.

Lemma 5.1 *π is a permutation.*

Proof: We first claim that every integer is present in the sequence. For the sake of argument assume that integer j is not present in the sequence π. If $a_j = 1$ and $a_{j+1} = 0$ then π_{j+1} is j. If $a_j = 0$ and $a_{j-1} = 1$ then π_{j-1} is j. In all other cases π_j is j itself. Thus every integer in the range [1...n] is present in the sequence.

No integer is repeated as the length of the sequence π is only n. Hence π is a permutation. ∎

Theorem 5.2 *In $O(1)$, time with n processors on an EREW PRAM, the problem of finding OR of n bits can be reduced to the problem of "detecting" presence of pattern 10 in a string of length $n + 1$.*

[1] In this chapter we will use the term "string" for binary string

Proof: Let the given input string be $X = x_1, x_2, \cdots, x_n$. Create another string $Y = y_1, y_2, \cdots, y_{n+1}$ where

$$y_i = \begin{cases} x_i & \text{if } 1 \leq i \leq n \\ 0 & \text{if } i = n+1 \end{cases}$$

OR of given input is TRUE iff there is atleast one 10 pattern in Y. ∎

Theorem 5.3 *In $O(1)$, time with $2n$ processors on an EREW PRAM, problem of finding Parity of n bits can be reduced to the problem of finding the parity of* the number *of* 10 *patterns in a string of length $2n$.*

Proof: Let the given input string be $X = x_1, x_2, \cdots, x_n$. Create another string $Y = y_1, y_2, \cdots, y_{2n}$ with $y_{2i-1} = x_i$ and $y_{2i} = 0$ for $1 \leq i \leq n$.
Parity of given input is EVEN if and only if the no. of 10 patterns in the string Y is EVEN (In fact the no. 1's in the string X is same as the number of 10 patterns in the string Y). Hence the result. ∎

Lemma 5.2 *There is a one-one relation between* 10 *patterns in the given string and the edges in the permutation graph of π constructed from the string.*

Proof: π_i can be either $i-1$, i, or $i+1$. Thus $\pi_i < \pi_{i+k}$, for $3 \leq k \leq n-i$. Further if π_i is $i+1$ then $\pi_{i+1} = i$; moreover as π is a permutation, π_{i+2} can not be $i+1$ Hence, we can say that $\pi_i < \pi_{i+k}$, for $2 \leq k \leq n-i$. This implies that there is no path of length 2 or more in the permutation graph. In other words any node i is connected with atmost one node— node $i-1$ or node $i+1$ (but not both).

Let $a_i a_{i+1}$ be an instance of 10 (i.e. $a_i = 1$ and $a_{i+1} = 0$). Then $\pi_i = i+1$ and $\pi_{i+1} = i$ hence there is an edge in the permutation graph between nodes i and $i+1$.

Conversely, if there is an edge between nodes j and $j+1$ in the permutation graph, then $\pi_j = j+1$ and $\pi_{j+1} = j$, or $a_j = 1$ and $a_{j+1} = 0$. Hence for every edge, there is a 10 pattern. ∎

Theorem 5.4 *In $O(1)$ time, problem of counting the number of* 10 *patterns can be reduced to the problem of finding the cardinality of maximum independent set of permutation graph.*

Proof: Consider the graph of permutation π constructed from the given string. Each vertex in the graph is either of degree 0 or 1. Let n_0 be the number of vertices of degree 0, and n_1 be the number of vertices of degree 1; $n_0 + n_1 = n$. Let e be the number of edges in the graph; $e = \frac{n_1}{2}$. Let $\mathcal{I}$ be a maximal independent set. Clearly all n_0 vertices of degree 0 will be in $\mathcal{I}$. As no vertex is of degree 2 or more, exactly one end-point of each edge will be in $\mathcal{I}$. If $CMIS$ is the cardinality of maximum independent set, then $CMIS = n_0 + e = n - e$.

But by Lemma 5.2 the number of 10 patterns in the graph is exactly equal to e. Thus, if we know the number of 10 patterns we know the cardinality of maximal independent set. ∎

As problem of finding "OR" of n-bits requires $\Omega(\log n)$ time on a Concurrent Read Exclusive Write PRAM [17], we get the following corollary.

Corollary 5.2 *The problem of finding cardinality of a maximum independent set of a permutation graph of n nodes requires $\Omega(\log n)$ time on a Concurrent Read Exclusive Write PRAM.*

As problem of finding "PARITY" of n-bits requires $\Omega(\frac{\log n}{\log\log n})$ on a Concurrent Read Concurrent Write PRAM [7], we get the following corollary.

Corollary 5.3 *The problem of finding cardinality of a maximum independent set of a permutation graph of n nodes requires $\Omega(\frac{\log n}{\log\log n})$ time on a Concurrent Read Concurrent Write PRAM, if only $n^{O(1)}$ processors are to be used.*

Remark: It is known that the complement of a permutation graph is also a permutation graph. In fact it can be shown that

$$G[\pi^r] = \overline{G[\pi]}$$

where π^r is the permutation obtained by reversing the sequence π and $\overline{G[\pi]}$ represents the complement of the graph $G[\pi]$. As the cardinality of maximum independent set of G equals the cardinality of maximum clique of $\overline{G}$, the lower bounds proved above apply to the problem of computing the cardinality of maximum clique of permutation graph.

Chapter 6

Permutation Representation and Coloring of a Cograph

Many graph problems like coloring, maximum independent set, minimum dominating set etc. are NP-Complete or intractable for general graphs [24]. Most of these problems have a polynomial time algorithm for special graphs like Interval graphs, Permutation graphs, Cographs and Chordal graphs (see [25] for definition).

Cographs are also called D*-graphs [30], P_4 restricted graphs [18] and Hereditary Dacey graphs [48]. Several characterizations are known for cographs, but perhaps, the most important characterization for algorithm design is its cotree representation.

6.1 Definitions and Preliminaries

DEFINITION: A *cograph* is defined as follows:

a) A graph consisting of a single vertex is a cograph

b) If $G_1, G_2, \ldots, G_n$ are cographs then their union $G_1 \cup G_2 \cup \ldots \cup G_n$ is a cograph

c) If G is a cograph then its complement $\overline{G}$ is also a cograph

DEFINITION: A *cotree* is a tree representing a cograph. The leaves of the cotree represents the vertices of the cograph. The internal nodes are numbered either 0 or 1. The root is numbered 1 only. If a node is numbered 1 then all of its children are numbered 0 and vice-versa. There is an edge between two nodes if and only if their lowest common ancestor

is numbered 1.

Cotree construction from the parse tree or definition tree is given in [28]. Every permutation graph can be represented by a permutation [25]. As the set of cographs is a proper subset of set of permutation graphs, a cograph can also be represented by a permutation.

6.2 Permutation Representation

A parallel algorithm for recognizing permutation graphs and constructing their permutation representation takes $O(\log^2 n)$ time with n^4 processors [29], where n is the number of nodes in the permutation graph. However, if cotree of a cograph is given then its permutation can be constructed in $O(\log n)$ time with n processors [28]. In this chapter, an optimal $O(\log n)$ time algorithm with $\frac{n}{\log n}$ processors is described.

Let $G = (V, E)$ be a cograph with cotree T. Let L_v be the set of leaves in the subtree rooted at v. The algorithm of [28] plots the vertices V of G on an $n \times n$ grid. For each node v of T, the vertices of G_v (the subgraph induced by vertices of L_v), are plotted in $\mid L_v \mid \times \mid L_v \mid$ square region on the grid. The coordinates of the lower left corner (x_v, y_v) and the upper right corner (x'_v, y'_v) can be calculated in an inductive manner.

If coordinates of v are known and if $v_1, v_2, \ldots, v_k$ are children of v then the corresponding coordinates of its children are computed as follows [28]:

Case 1— v is a 0 node See Figure 6.1(a):

$$x_{v_i} = x_v + \sum_{j=1}^{i-1} \mid L_{v_j} \mid \tag{6.1}$$

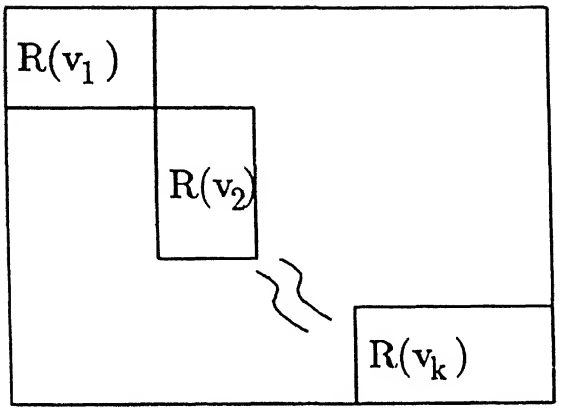
$$y_{v_i} = y_v + \sum_{j=1}^{i-1} \mid L_{v_j} \mid \tag{6.2}$$

$$x'_{v_i} = x_v + \sum_{j=1}^{i} \mid L_{v_j} \mid -1 \tag{6.3}$$

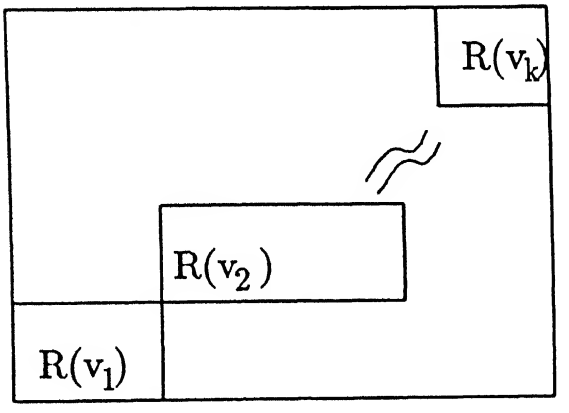
$$y'_{v_i} = y_v + \sum_{j=1}^{i} \mid L_{v_j} \mid -1 \tag{6.4}$$

Case 2— v is a 1 node See Figure 6.1(b):

$$x_{v_i} = x_v + \sum_{j=1}^{i-1} \mid L_{v_j} \mid \tag{6.5}$$



(a)



(b)

Figure 6.1: Computing regions for children

$$y_{v_i} = y_v + \sum_{j=i-1}^{k} | L_{v_j} | \tag{6.6}$$

$$x'_{v_i} = x_v + \sum_{j=1}^{i} | L_{v_j} | -1 \tag{6.7}$$

$$y'_{v_i} = y_v + \sum_{j=i}^{k} | L_{v_j} | -1 \tag{6.8}$$

For a leaf node w, the lower-left-corner and the upper-right-corner coincide; i.e. $x_w = x'_w$ and $y_w = y'_w$. The permutation representation for the cograph, π can be obtained as follows: for each leaf w let $\pi_{x_w} = y_w$.

The recursive equations $6.1, 6.2, \ldots, 6.8$ can be solved in parallel as follows:

1) For node v with children $v_1, v_2, \ldots, v_k$ compute the prefix and the postfix summation of the corresponding $| L_{v_i} |$'s.
2) Carry out prefix computation on the cotree (see Chapter 4 for details).

As both these steps can be implemented in $O(\log n)$ time with $(\frac{n}{\log n})$ processors on an EREW PRAM, the following theorem holds:

Theorem 6.1 *Permutation representation of a cograph, given its cotree representation, can be obtained in $O(\log n)$ time with $(\frac{n}{\log n})$ processors on an EREW PRAM.* ∎

6.3 Coloring

Algorithms for maximum weighted clique and minimum coloring of a permutation graph of [2]. take $O(\log^2)$ with n^3 processors. However, the maximum clique of a cograph (given its cotree) can be obtained in $O(\log n)$ time using $\frac{n}{\log n}$ processors on an Exclusive Read and Exclusive Write PRAM [1]. The algorithm given in [1] can be extended to find a maximum weighted clique; as cograph is a perfect graph, the chromatic number is equal to its clique number. We obtain an algorithm for minimum coloring with the same resourse bounds.

The algorithm of [1] is first described for completeness; later the algorithm is extended to obtain a minimum coloring. Let G_v be the subgraph induced by the vertices of L_v. And let $C(v)$ be the chromatic (or clique) number of G_v. The chromatic number of each vertex in a cotree can be calculated as follows:

Case 1— v is a leaf node: chromatic number of single vertex graph is one, hence $C(v)$ is equal to one in this case.

Case 2— v is numbered 0: Let v has k children, namely $v_1, v_2, \ldots, v_k$. As v is numbered 0, G_v is disjoint union of $G_{v_1}, G_{v_2}, \ldots, G_{v_k}$. Hence $C(v)$ will be $\max(C(v_1), C(v_2), \ldots, C(v_k$

Case 3— v is numbered 1: Let v has k children, namely $v_1, v_2, \ldots, v_k$. As v is numbered 1, in G_v there will be all possible edges between subgraphs $G_{v_1}, G_{v_2}, \ldots, G_{v_k}$. Hence chromatic number of $C(v)$ will be summation of chromatic number of its children.

Theorem 6.2 ([1]) *Chromatic number of a cograph can be found in $O(\log n)$ time using $\frac{n}{\log n}$ processors on an EREW PRAM model.* ∎

If $C(v)$ is m then we need m colors to color the subgraph G_v. Let us say we color G_v with colors $[i \ldots \subset +m-1]$; we say that the range of node v is $[i \ldots i+m-1]$. Let the chromatic number of root, r be c, i.e. the given cograph is colorable with c colors. Let the root's children be $v_1, v_2, \ldots, v_k$ with $c_1, c_2, \ldots, c_k$ as their chromatic numbers respectively. As root is numbered 1, we know that $c = c_1 + c_2 + \cdots + c_k$. The ranges of each child of root is computed as below:

range of v_1 will be $[1 \ldots c_1]$
range of v_2 will be $[1 + c_1 \ldots c_1 + c_2]$
range of v_i will be $[1 + \sum_{j=1}^{i-1} c_j \cdots \sum_{j=1}^{i} c_j]$, for $3 \leq i \leq k$

The above formulae are true for any node numbered 1 except that instead of 1 in the formulae it should be its lower value. Let the range of a node v be $p \ldots q$. the ranges of children of v will be calculated as follows.

range of v_1 will be $[p \ldots c_1]$
range of v_2 will be $[p + c_1 \ldots c_1 + c_2]$
range of v_i will be $[p + \sum_{j=1}^{i-1} c_j \cdots \sum_{j=1}^{i} c_j]$, for $3 \leq i \leq k$

If the node is numbered 0 then its range can be carried to all of its children.

For a leaf node the upper and lower values of the range coincide. In fact the upper value of any node can be $p + C(v) - 1$, where p is its lower value and $C(v)$ is its chromatic number. Hence propagation of lower value is enough.

Above procedure can be implemented efficiently as follows:

1) Find $C(v)$, the chromatic number of each v in the cotree.

2)Carry out the following sub steps:

a) Let for each node v numbered 0, $\beta(v) = \sum_i C(b_i) - C(v)$ where b_i is a right brother[1] of v.

b) Let for each node v numbered 1, $\beta(v) := 0$

3)Apply prefix computation on trees (see Chapter 4) for $\beta(v)$ with '+' as the associative binary operator.

Hence the following theorem follows,

Theorem 6.3 *Minimum coloring of a cograph can be done in* $O(\log n)$ *time using* $(\frac{n}{\log n})$ *processors on an EREW PRAM model if its cotree is given.* ∎

[1] If $v_1, v_2, \ldots, v_k$ are children of a node v in that order, we say, $v_1, v_2, \ldots, v_i$ are right brothers of node v_i

Chapter 7

Conclusions and Scope for Further Work

In this thesis efficient parallel algorithms have been obtained for some of the interesting problems on Special graphs like Interval graph, Permutation graph, Cograph etc. It is possible to design better algorithms on these graphs because they possess some special properties which can be used to obain better algorithms. An interesting problem not considered in this thesis is that of recognition of these graphs.

Test for connectivity of an interval graph, given its interval model, takes at least $O(n \log n)$ time where n is the number of intervals [39]. However, if the intervals were given in sorted order then it is possible to design even doubly-logarithmic time and even faster algorithms using k-th-prefix-maxima technique. It may be possible to use this technique to obtain better algorithms for other problems also.

However, this technique does not work for k edge connected components. To the best of our knowledge there is no known algorithm for finding k edge connected components of an interval graph. Finding such an algorithm is an interesting open problem.

The best algorithm (sequential) for finding maximum weighted independent set of a permutation graph takes $O(n \log\log n)$ time [52]. The fastest parallel algorithm takes $O(\log^2)$ time [52] and is far from being optimal. The problems of finding optimal $O(\log n)$ time parallel algorithms for maximum weighted independent set and minimum weighted dominating set are open.

Appendix A

Strong Perfect Graph Conjecture

A.1 p-Critical Graphs

An undirected graph $G(V, E)$ is called *p-critical* if $G(V, E)$ is not perfect but every proper induced subgraph of G is perfect. In other words removal of any vertex makes the graph perfect. More formally, for any node v the following are true:

$$\omega(G - v) = \chi(G - v) \tag{A.1}$$

$$\alpha(G - v) = \kappa(G - v) \tag{A.2}$$

where $G - v$ represents the subgraph induced by all vertices of V but v.

Theorem A.1 (see [25]) *If $G(V, E)$ is a p-critical graph, then $\alpha(G)\omega(G) + 1 = \mid V \mid$ and for all vertices v of G, $\alpha(G) = \kappa(G - v)$ and $\omega(G) = \chi(G - v)$.*

The odd cycle C_{2k+1}[1], where $k \geq 2$, is not a perfect graph since its clique number is 2 and its chromatic number is 3. But removal of any node from C_{2k+1} results in a perfect graph or in other words every proper subgraph of C_{2k+1} is perfect. Its complement $\overline{C}_{2k+1}$ is also p-critical. In fact, C_{2k+1} and $\overline{C}_{2k+1}$ (where $k \geq 2$) are the only p-critical graphs known to date.

Berge conjectured that there are no other p-critical graphs. This is known as *Strong Perfect Graph Conjecture* (SPGC).

[1] C_{2k+1} is a connected graph of $2k + 1$ vertices in which every vertex degree is 2.

A.2 Equivalent forms of SPGC

SPGC-1: An undirected graph G is perfect iff it contains no induced subgraph isomorphic to C_{2k+1} or $\overline{C}_{2k+1}$ where $k \geq 2$.

SPGC-2: The only p-critical graphs that exist are C_{2k+1} or $\overline{C}_{2k+1}$ where $k \geq 2$.

SPGC-3: there is no p-critical graph with $\alpha(G) > 2$ and $\omega(G) > 2$.

If we restrict the universe of graphs being considered, then in some cases the SPGC can be shown to hold. It is proved that the SPGC holds for the following problems:

- Planar graphs [49].
- $K_{1,3}$-free graphs [40].
- Graphs whose chromatic number is less than or equal to 3 [50].
- Toroidal graphs; graphs having maximum vertex degree lee than or equal to 6 [26].

Bibliography

[1] Abrahamson, Dadoun, Kirkpatrick and Przytycka. A simple parallel tree contraction algorithm. *Journal of Algorithms*, 10(1):287–302, 1989.

[2] Adhar G. S. and Peng S. T. Parallel Algorithms for Cographs and Parity Graphs with Applications. *Journal of Algorithms*, 11(2):252–284, 1990.

[3] Aho A., Hopcroft J. and Ullman J. *The Design and Analysis of Computer Algorithms.* Addison-Wesley, 1974.

[4] Albers, S., Hagerup,T. Improved Parallel Integer Sorting without Concurrent Writing. In *Proc 3rd Ann. ACM-SIAM Symp on Discrete Alg.*, pages 463–472, 1992.

[5] Alt H., Hagerup T., Melhorn K., and Preparata F. Deterministic simulation of idealised parallel computers on more realistic ones. In *proc. Mathematical Foundations of Computer Science, Lec. Notes in Comp. Science, Vol. 233.* Springer-Verlag, 1986.

[6] Anderson R. J. and Miller G. L. Deterministic Parallel List Ranking. *Algorithmica*, 6(6):859–868, 1991.

[7] Beame, P., Hastad, J. Optimal Bounds for Decision Problems on the CRCW PRAM. *Journal of ACM*, 36(3):643–670, 1989.

[8] Berkman O. Paradigms for Very Fast Parallel Algorithms, Ph. D. thesis, Tel Aviv University. UMIACS-TR-91-117, University of Maryland, August 1991.

[9] Berkman O., Ja'Ja' J., Krishnamurthy S., Thurimella R. and Vishkin U. Some Triply-Logarithmic Parallel Algorithms. In *Proceedings IEEE FOCS*, pages 871–881, 1990.

[10] Berkman O., Schieber B. and Vishkin U. Optimal Doubly Logarithmic Parallel Algorithms Based on Finding all Nearest Smaller Values. *Journal of Algorithms*, 14(3):344–370, 1993.

[11] Bertossi A. A. and Bonuccelli M. A. Some Parallel Algorithms on Interval Graphs. *Discrete Applied Mathematics*, 16:101–111, 1987.

[12] Bhatt,P.C.P., Diks,K., Hagerup,T., Prasad,V.C., Radzik,T., Saxena,S. Improved Deterministic Parallel Integer Sorting. *Information and Computation*, 94:29–47, 1991.

[13] Cherian J., Kao M. Y. and Thurimella R. 'Scan first Search and Sparse certificates: An improved parallel algorithm for k-vertex connectivity. *SIAM Journal of Computing*, 22(1):157–174, 1993.

[14] Chiang Y. and Fu K. S. Parallel processing for distance computation in syntactic pattern recognition. In *Proceedings of IEEE Computer Society Workshop on Computer Architecture for Pattern Analysis and Image database management*, 1981.

[15] Cole R. and Vishkin U. Deterministic Coin Tossing with Applications to Optimal Parallel List Ranking. *Information and Control*, 70(1):32–53, 1986.

[16] Cole R. and Vishkin U. Approximate parallel scheduling part I: the basic technique with applications to optimal parallel list ranking in logarithmic time. *SIAM Journal of Computing*, 17(1):128–142, 1988.

[17] Cook S., Dwork C. and Reischuk R. Upper and lower time bounds for Parallel Random Access Machines without Simultaneous writes. *SIAM Journal of Computing*, 15(1):87–97, 1986.

[18] D. G. Corneil, Y. Perl and L. K. Stewart. A Linear Recognition Algorithm for Cographs. *SIAM Journal of Computing*, 14(4):926–934, 1985.

[19] Das S. K., Calvin and Chen C. Y. Efficient Parallel Algorithms on Interval Graphs. In *PARLE 92, Lec. Notes in Comp. Science, Vol. 605*, pages 131–141. Springer-Verlag, 1992.

[20] Deo N. *Graph Theory with Applications to Engineering and Computer Science.* Series in Automatic Computation. Prentice-Hall, 1974.

[21] Even S., Pnueli A. and Lempel A. Transitive Orientation of graphs and identification of Permutation Graphs. *Canadian Journal of Mathematics*, 23:160–175, 1971.

[22] Even S., Pnueli A. and Lempel A. Permutation graphs and Transitive graphs. *Journal of ACM*, 19(3):400–410, 1972.

[23] Faith E., Prabhakar R. and Wigderson A. Relations between concurrent-write models of computation. *SIAM Journal of Computing*, 17(3):606–627, 1988.

[24] Garey M. R. and Johnson D. S. *Computers and Intractability: A Guide to the Theory of NP-completeness.* Freeman, San Francisco, 1979.

[25] Golumbic M. C. *Algorithmic Graph Theory and Perfect Graphs.* Computer Science and Applied Mathematics. Academic Press, New York, 1980.

[26] Grindstead C. M. *Toroidal graphs and the strong perfect graph conjecture.* PhD thesis, University of California, Los Angeles, 1978.

[27] Harary F. *Graph Theory.* Addison-Wesley, 1968.

[28] He X. Parallel Algorithm for Cograph Recognition with Applications. *Journal of Algorithms*, 15(2):284–313, 1993.

[29] Helmbold D. and Mayr E. Perfect graphs and parallel algorithms. In *Proceedings, International Conference on Parallel Processing*, pages 853–860, 1986.

[30] Jung H. A. On a Class of Posets and the Corresponding Comparability Graphs. *Journal of Combinatorial Theory, Series B*, 24(2):125–133, 1978.

[31] Kim H. Finding maximum independent set in a permutation graph. *Information Processing Letters*, 36(1):19–23, 1990.

[32] Knuth D. E. *The Art of Computer Programming*, volume 1. Addison-Wesley, 1973.

[33] Kruskal C. P., Rudolph L. and Snir M. The power of Parallel Prefix. *IEEE Transactions on Computers*, C-34(10):965–968, 1985.

[34] Liang Y., Dhall S. K. and Lakshmivarahan S. Efficient Parallel Algorithms for Finding Biconnected Components of Some Intersection Graphs. In *ACM Computer Science Conference*, pages 48–52, 1991.

[35] Lovász. A characterization of perfect graphs. *Journal on Combinatorial Theory, Series B*, 13:95–98, 1972.

[36] Lovász. Normal hyper graphs and the perfect graph conjecture. *Discrete Mathematics*, 2:253–267, 1972.

[37] Lu Mi. Parallel Computation of Longest-Common-Subsequence. In *Advances in Computing and Information, Lec. Notes in Comp. Science, Vol. 468*, pages 385–394. Springer-Verlag, 1990.

[38] Modelevsky J. L. Computer applications in applied genetic engineering. *Advances in Microbiology*, 30:169–195, 1984.

[39] Olariu S., Schwing J. L. and Zhang J. Optimal Parallel Algorithms for Problems Modeled by a Family of Intervals. *IEEE transactions on Parallel and Distributed Systems*, 3(3):364–374, 1992.

[40] Parthasarathy K. R. and Ravindra G. The strong perfect graph conjecture is true for $k_{1,3}$ -free graphs. *Journal of Combinatorial TIleory, series B*, 21:212–223, 1976.

[41] Prabhakar R. The parallel Simplicity of Compaction and Chaining. *Journal of Algorithms*, 14(3):371–380, 1993.

[42] Ramkumar G. D. S. and Rangan C. P. Parallel Algorithms on Interval Graphs. In *Proc. Int. Conf. Parallel process.*, pages 72–74, 1990.

[43] Saxena S. *Lecture Notes for course CS742, 1992-1993, Second Semester.* Dept. of computer science and engineering, IIT Kanpur.

[44] Spinrad J. On Comparability and Permutation Graph. *SIAM Journal of Computing*, 14(3):658–670, 1985.

[45] Sprague A. P., and Kulkarni K. H. Optimal parallel algorithms for finding cut vertices and bridges of interval graphs. *Information Processing Letters*, 42(4):229–234, 1992.

[46] Springsteel and Stojmenovic. General parallel prefix computation with geometric, algebraic, and other applications. *International Journal of Parallel Programming*, 18(6):485-503, 1989.

[47] Sridhar M. A. and Goyal S. Efficient Parallel Computation of Hamiltonian Paths and Circuits in Interval Graphs. In *Proc. Int. Conf. Parallel process.*, pages 83–90, 1991.

[48] Sumner D. P. Dacey graphs. *J. Austral. Math. Soc.*, 18(4):492–502, 1974.

[49] Tucker A. C. The strong perfect graph conjecture for planar graphs. *Canadian Journal of Mathematics*, 25:103-114, 1973.

[50] Tucker A. C. Critical perfect graphs and perfect 3-chromatic graphs. *Journal of Combinatorial Theory, series B*, 23:143-149, 1977.

[51] Wagner H. Triangulating a monotone polygon in parallel. In *Computational Geometry and Its Applications, Lec. Notes in Comp. Science, Vol. 333*, pages 136–147. Springer-Verlag, 1988.

[52] Yu M. S., Tseng L. Y. and Chang. Sequential and parallel algorithms for the maximum-weight independent set problem on permutation graphs. *Information Processing Letters*, 46(1):7–11, 1993.